Polyhedral-Model Guided Loop-Nest Auto-Vectorization

Konrad Trifunović, Dorit Nuzman, Albert Cohen, Ayal Zaks, Ira Rosen

Polyhedral-Model Guided Loop-Nest Auto-Vectorization

Konrad Trifunovic †, Dorit Nuzman *, Albert Cohen †, Ayal Zaks * and Ira Rosen *
*IBM Haifa Research Lab, {dorit, zaks, irar}@il.ibm.com
†INRIA Saclay, {konrad.trifunovic, albert.cohen }@inria.fr

***Abstract*—Optimizing compilers apply numerous inter-dependent optimizations, leading to the notoriously difficult *phase-ordering* problem — that of deciding which transformations to apply and in which order. Fortunately, new infrastructures such as the *polyhedral* compilation framework host a variety of transformations, facilitating the efficient exploration and configuration of multiple transformation sequences. Many powerful optimizations, however, remain external to the polyhedral framework, including vectorization. The low-level, target-specific aspects of vectorization for fine-grain SIMD has so far excluded it from being part of the polyhedral framework.**

In this paper we examine the interactions between loop transformations of the polyhedral framework and subsequent vectorization. We model the performance impact of the different loop transformations and vectorization strategies, and then show how this cost model can be integrated seamlessly into the polyhedral representation. This predictive modelling facilitates efficient exploration and educated decision making to best apply various polyhedral loop transformations while considering the subsequent effects of different vectorization schemes. Our work demonstrates the feasibility and benefit of tuning the polyhedral model in the context of vectorization. Experimental results confirm that our model has accurate predictions, providing speedups of over 2.0x on average over traditional innermost-loop vectorization on PowerPC970 and Cell-SPU SIMD platforms.

I. Introduction

Fine-grain data level parallelism is one of the most effective ways to achieve scalable performance of numerical computations. Automatic vectorization for modern short-SIMD instruction sets, such as Altivec, Cell SPU and SSE, has been a popular topic, with successful impact on production compilers [1], [2], [3], [4]. Exploiting subword parallelism in modern SIMD architectures, however suffers from several limitations and overheads (involving alignment, redundant loads and stores, support for reductions and more) which complicate the optimization dramatically. Automatic vectorization was also extended to handle more sophisticated control-flow restructuring including if-conversion [5] and outer-loop vectorization [6]. Classical techniques of loop distribution and loop interchange [7] can dramatically impact the profitability of vectorization. To be successful, it is vital to avoid inapt strategies that incur severe overheads. Nevertheless, little has been done to devise reliable profit models to guide the compiler through this wealth of loop nest transformation candidates, vectorization strategies and code generation techniques. Our main goal in this paper is to propose a practical framework for such guidance.

Modern architectures must exploit multiple forms of parallelism provided by platforms while using the memory hierarchy efficiently. Systematic solutions to harness the interplay of multi-level parallelism and locality are emerging, by advances in automatic parallelization and loop nest optimization [8], [9]. These rely on the polyhedral model of compilation to facilitate efficient exploration and application of very complex transformation sequences. However, exploiting subword parallelism by vectorization is excluded from the polyhedral model due to its low-level machine-dependent nature. As a result, there remains a gap in providing a combined framework for exploring complex loop transformation sequences together with vectorization. In this work we help bridge this gap by incorporating vectorization considerations into a polyhedral model. This methodology could be extended in the future to consider the effects of additional transformations within the polyhedral framework. The contributions of this work are fourfold:

- **Cost model for vectorization.** We developed a fast and accurate cost model designed to compare the performance of various vectorization alternatives and their interactions with other loop optimizations.
- **Polyhedral modelling of subword parallelism.** We demonstrate how to leverage the polyhedral compilation framework naturally and efficiently to assess opportunities for subword parallelism in combination with complex loop transformation sequences.
- **Evaluation in a production compiler.** Our model is fully automated and implemented based on GCC 4.4.
- **Studying the interplay between loop transformations.** We provide a thorough empirical investigation of the interplay between loop interchange with array expansion and loop nest vectorization of both inner and outer loops on modern short-SIMD architectures.

The rest of this paper is organized as follows: Section II discusses related work. Section III studies a motivating example. Section IV provides an overview, introduces some notation and captures loop interchange and vectorization variants as affine transformations. Section V describes the optimization process in detail. Section VI describes the cost function. Section VII exposes performance results, and Section VIII concludes.

II. Related Work

Vectorization Cost-Model Related Work. Leading optimizing compilers recognize the importance of devising a cost model for vectorization, but have so far provided only partial solutions. Wu et al. conclude [1] regarding the XL compiler that *"Many further issues need to be investigated before we can enjoy the performance benefit of simdization ... The more important features among them are ... the ability to decide when simdization is profitable. Equally important is a better understanding of the interaction between simdization and other optimizations in a compiler framework"*. Likewise, Bik stresses the importance of user hints in the ICC vectorizer's profitability estimation [2], to avoid vectorization slowdowns due to *"the performance penalties of data rearrangement instructions, misaligned memory references, failure of store-to-load forwarding, or additional overhead of run-time optimizations to enable vectorization."*; on the other hand opportunities may be missed due to overly conservative heuristics.

These state-of-the-art vectorizing compilers incorporate a cost model to decide whether vectorization is expected to be profitable. These models however typically apply to a single loop or basic-block, and do not consider alternatives combined with other transformations at the loop-nest level. This work is the first to incorporate a polyhedral model to consider the overall cost of different vectorization alternatives in a loop-nest, as well as the interplay with other loop transformations.

Loop-nest auto-vectorization in conjunction with loop-interchange has been addressed in prior art [10], [7], [11]. This however was typically in the context of traditional vector machines (such as Cray), and interchange was employed as a preprocessing enabling transformation. Overheads related to short-SIMD architectures (such as alignment and fine-grained reuse) were not considered.

Costs of specific aspects of short-SIMD vectorization were addressed in more recent works. Realignment and data-reuse were considered together with loop-unrolling [12], but in the context of straight-line code vectorization, and not for the purpose of driving loop vectorization. A cost model for vectorization of strided-accesses was proposed in [13], but it does not consider other overheads or loop transformations.

Polyhedral-Model Related Work. Bondhugula et al. [8] integrate inner-loop vectorization as a post-pass of their tiling heuristic, and leverage the interchangeability of inner loops to select one that is vectorizable. Their method does not take into consideration the respective vectorization overheads, nor does it model reductions. Nevertheless, their tiling hyperplane and fusion algorithm can serve as a complementary first pass for our technique, favoring the extraction of interchangeable inner loops.

Pouchet et al. demonstrate how one can systematically study the interplay of loop transformations with backend optimizations (including vectorization) and complex microarchitectures by constructing huge search spaces of unique, valid transformation sequences [9]. These search spaces are tractable using carefully crafted heuristics that exploit the structure of affine schedules. An analytical performance model capable of characterizing the effect of such complex transformations (beyond loop interchange, and accommodating for large-scale locality effects) does not currently exist. Known analytical cache models for loop transformations are quite mature in some domains, loop tiling in particular [14], yet remain sensitive to syntactic patterns and miss key semantical features such as loop fusion effects [15], [16].

III. Motivating Example

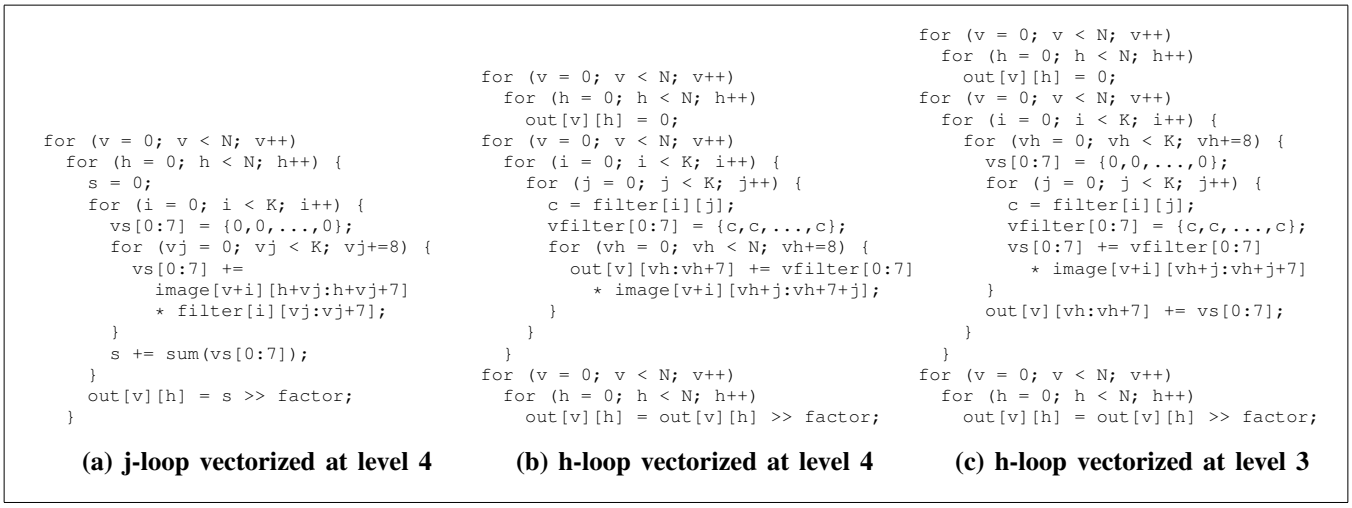
```
   for (v = 0; v < N; v++)
     for (h = 0; h < N; h++) {
S1:    s = 0;
       for (i = 0; i < K; i++)
         for (j = 0; j < K; j++)
S2:        s += image[v+i][h+j] * filter[i][j];
S3:    out[v][h] = s >> factor;}
```

Figure 1. Main loop kernel in Convolve

The first and foremost goal of a vectorization cost-model is to avoid performance degradations while not missing out on improvement opportunities. In addition, a cost model should also drive the selection of a vectorization strategy, assuming there exist a profitable one. Given a loop-nest, a compiler needs to choose which loop to vectorize, and at which position, employing one of several strategies (innermost- or outer-loop vectorization, in-place or based on innermosting, as explained below). This in-turn brings to play other loop-transformations, most notably loop-interchange but also loop-peeling and others. Searching among all these alternatives becomes a non-trivial problem. This problem is especially apparent in computations featuring deep loop nests that can be vectorized in several ways, and are amenable to other loop optimizations.

Figure 1 introduces the *Convolve* kernel – a simple example of a loop nest exhibiting the above features. We use this example in the rest of the paper to demonstrate our techniques.

There are d possible vectorization alternatives for a loop-nest of depth d (in our case $d = 4$), without involving any other loop-transformation: we can vectorize any of the j, i, h or v loops "in-place" – i.e. in their original position (loop-level). For instance, we can vectorize the j-loop in its innermost position (as shown in Figure 2a), which is the common practice of vectorizing compilers. Employing loop-interchange to permute one loop (inwards or outwards) into a specific position within the loop nest and vectorizing it there (keeping the other loops intact), increases the search-space to $d \cdot d$ possibilities. Figures 2b and 2c show examples



```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++) {
    s = 0;
    for (i = 0; i < K; i++) {
      vs[0:7] = {0,0,...,0};
      for (vj = 0; vj < K; vj+=8) {
        vs[0:7] +=
          image[v+i][h+vj:h+vj+7]
          * filter[i][vj:vj+7];
      }
      s += sum(vs[0:7]);
    }
    out[v][h] = s >> factor;
  }
```

(a) j-loop vectorized at level 4

```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = 0;
for (v = 0; v < N; v++)
  for (i = 0; i < K; i++) {
    for (j = 0; j < K; j++) {
      c = filter[i][j];
      vfilter[0:7] = {c,c,...,c};
      for (vh = 0; vh < N; vh+=8) {
        out[v][vh:vh+7] += vfilter[0:7]
          * image[v+i][vh+j:vh+7+j];
      }
    }
  }
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = out[v][h] >> factor;
```

(b) h-loop vectorized at level 4

```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = 0;
for (v = 0; v < N; v++)
  for (i = 0; i < K; i++) {
    for (vh = 0; vh < K; vh+=8) {
      vs[0:7] = {0,0,...,0};
      for (j = 0; j < K; j++) {
        c = filter[i][j];
        vfilter[0:7] = {c,c,...,c};
        vs[0:7] += vfilter[0:7]
          * image[v+i][vh+j:vh+j+7]
      }
      out[v][vh:vh+7] += vs[0:7];
    }
  }
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = out[v][h] >> factor;
```

(c) h-loop vectorized at level 3

Figure 2. Convolve Vectorization Examples

of vectorizing the h-loop after permuting it to the innermost and next-to-innermost positions, respectively. Using loop-permutation more aggressively to reorder the loops of a nest according to a specific permutation, and then vectorizing one of them, results in a total of $d(d!)$ combinations. If we also employ loop peeling to align memory accesses, the search space grows to $d(d!)VF$ where VF is the Vectorization Factor (number of elements operated upon in parallel in a vector). In our case this amounts to 768 alternatives.

The search space becomes quite large even for modest depths and only few transformations (interchange and peeling), as shown, and can easily reach much higher volumes if deeper loop nests and/or more loop-transformations are considered. Also note that programs often contain many loop-nests, where each loop-nest should be optimized.

Approaches that generate each alternative and rely on its (possibly simulated) execution or on performance evaluation at a later low-level compilation stage, are competitive in terms of accuracy but are significantly inferior in terms of scalability to analytical approaches that reason about costs and benefits without actually carrying out the different loop transformations beforehand. Operating on the polyhedral representation itself, rather than relying on code generation, is therefore a key ingredient. Hybrid approaches can provide a more practical solution by combining the feedback-based approach with classical analytical models to narrow the search space. The modest compile-time requirements of our purely analytical approach (about 0.01s to build the model and search for the optimal vectorization strategy for *Convolve*) facilitates its integration in a production compiler.

A complementary challenge to dealing with the very large search-spaces, is how to evaluate the costs and benefits associated with each alternative efficiently and accurately. Some tradeoffs are clearly visible in Figure 2. For example, variants (b,c) use loop-permutation, which in this case incurs an overhead of extra memory traffic to/from the out array. On the other hand variant (a) incurs a reduction epilog overhead (see sum operation) in each iteration of the i-loop. Outer-loop vectorization (vectorizing a loop other that innermost-loop) is used in (c), implying that more code is vectorized. The innermost j-loop in this case continues to advance sequentially, operating simultaneously on values from $VF = 8$ consecutive h-loop iterations. On the other hand (b) has better temporal locality ($filter[i][j]$ is invariant in the innermost loop) and the misalignment is fixed (this is explained in more detail later). Overall the speedup factors obtained by transformations a, b, c on PPC970 (relative to the original sequential version shown in Figure 1) are 2.99, 3.94, 3.08 respectively. On the Cell SPU the respective speedups are 2.59, 1.44, 3.62.

The following sections describe our approach and demonstrate how our cost model computes its predictions within the analytical polyhedral-based model, considering different loop transformations and metrics. Final cost-model predictions for *Convolve* and analysis of the speedups are given in Section VII-A, where we show that the cost model is able to correctly predict the best vectorization option for both PPC and SPU.

IV. Background and Notation

Most compiler internal representations match the inductive semantics of imperative programs including syntax tree, call tree, control-flow graph and SSA. In such reduced representations of the dynamic execution trace, each statement of a high-level program occurs only once, even if it is executed many times (e.g., when enclosed within a loop). Representing a program this way is not convenient for aggressive loop optimizations which operate at the level of dynamic *statement instances*.

Compile-time constraints and lack of adequate algebraic representation of loop nest semantics prevent traditional compilers from adapting the schedule of statement instances of a program to best exploit architecture resources. For example, compilers typically cannot apply loop transformations if data dependences are non-uniform [11] or simply because profitability is too unpredictable.

A. Polyhedral Compilation

A well known alternative approach, facilitating complex loop transformations, represents programs in the *polyhedral model*. This model is a flexible and expressive representation for loop nests with statically predictable control flow. Such loop nests, amenable to algebraic representation, are called *static control parts* (SCoP) [17], [18]; their control and data flow are split into three components:

1. *Iteration domains* capture the dynamic instances of all statements — all possible values of surrounding loop iterators — through a set of affine inequalities. Each dynamic instance of a statement S is denoted by a pair $(S, \mathbf{i})$ where $\mathbf{i}$ is the iteration vector containing values for the loop indices of the surrounding loops, from outermost to innermost. The dimensionality of iteration vector $\mathbf{i}$ is d^S. If loop bounds are affine expressions of outer loop indices and global parameters (usually, symbolic constants representing problem size) then the set of all iteration vectors $\mathbf{i}$ relevant for statement S can be represented by a polytope $\mathcal{D}^S = \left\{\mathbf{i} \mid \mathrm{D}^S \times (\mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$ which is called the *iteration domain* of the statement S, where $\mathbf{g}$ is the vector of *global parameters* whose dimensionality is $d_{\mathbf{g}}$.

For example, the domain $\mathcal{D}^{S_2}$ of the statement S_2 in Figure 1 has the following iteration domain representation:

$$\mathcal{D}^{S_2} = \{(v, h, i, j) \mid 0 \leq v, h \leq N-1 \wedge 0 \leq i, j \leq K-1\}$$

2. *Memory access functions* capture the locations of data on which statements operate. In static control parts, memory accesses are performed through array references. For each statement S we define two sets — $\mathcal{W}^S$ and $\mathcal{R}^S$ — of $(\mathtt{M}, f)$ pairs. Each pair represents a reference to a variable `M` being written or read by the statement S and f is the *access function* mapping iteration vectors in $\mathcal{D}^S$ to the memory locations in `M`. The access function f, defined by a matrix F such that:

$$f(\mathbf{i}, \mathbf{g}) = \mathrm{F} \times (\mathbf{i}, \mathbf{g}, 1)^T \tag{1}$$

is a vector valued function whose dimensionality is equal to that of array `M`. Statement S_2 in Figure 1 has the following access function sets:

$$\mathcal{W}^{S_2} = \emptyset, \mathcal{R}^{S_2} = \left\{ \begin{array}{l|l} \left(\mathtt{image}, \left[\begin{array}{c|c|c} 1010 & 00 & 0 \\ 0101 & 00 & 0 \end{array}\right]\right) & \begin{bmatrix} v+i \\ h+j \end{bmatrix}, \\ \left(\mathtt{filter}, \left[\begin{array}{c|c|c} 0010 & 00 & 0 \\ 0001 & 00 & 0 \end{array}\right]\right) & \begin{bmatrix} i \\ j \end{bmatrix} \end{array} \right\}$$

3. *Scheduling function*. Iteration domains define the set of dynamically executed instances of each statement. However, this algebraic structure does not describe the order in which each statement instance has to be executed with respect to other statement instances [9]. We should not rely on the inductive semantics of the sequence and loop iteration for this purpose, of course, as that would break the algebraic reasoning about loop nests.

A convenient way to express execution order is by giving each statement instance an execution date. It is obviously impractical to define all dynamic instances explicitly. An appropriate solution is to define a *scheduling function* θ^S for each statement S which maps instances of S to totally ordered multidimensional timestamps (vectors), explained in the next subsection. For tractability reasons, we restrict these functions to be affine.

B. Polyhedral Transformations

Each loop-nest transformation in the polyhedral model is represented as a schedule transformation or as a domain transformation. All transformations are applied statement-wise.

We define the multidimensional scheduling function as an affine form of the outer loop iterators $\mathbf{i}$ and the global parameters $\mathbf{g}$:

$$\theta^S(\mathbf{i}) = \Theta^S \times (\mathbf{i}, \mathbf{g}, 1)^T \tag{2}$$

where Θ^S is the *scheduling matrix* of constant integers. Statement instance $(S_i, \mathbf{i}^{S_i})$ executes before statement instance $(S_j, \mathbf{i}^{S_j})$ if and only if $\theta^{S_i}(\mathbf{i}^{S_i}) \ll \theta^{S_j}(\mathbf{i}^{S_j})$, where $\ll$ denotes the lexicographic order of schedule vectors. Every static control part has a multidimensional affine schedule [19]. By providing different scheduling functions for individual statements we can perform affine-by-statement transformations, improving over unimodular and classical syntax-tree based transformations [19], [20], [21], [18]. Efficient algorithms and tools exist to regenerate code from a polyhedral representation according to (modified) multidimensional affine schedules [22], [18].

Scheduling encodings using $2d^S + 1$ positions were previously proposed by Feautrier [19] and later by Pugh and Kelly [20]. These encodings were generalized to handle arbitrary compositions of affine transformations by Girbal et al. [18], using the following format of scheduling matrices:

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ \mathrm{A}_{1,1}^S & \cdots & \mathrm{A}_{1,d^S}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,d_{\mathbf{g}}}^S & \Gamma_{1,d_{\mathbf{g}}+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ \mathrm{A}_{2,1}^S & \cdots & \mathrm{A}_{2,d^S}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,d_{\mathbf{g}}}^S & \Gamma_{2,d_{\mathbf{g}}+1}^S \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathrm{A}_{d^S,1}^S & \cdots & \mathrm{A}_{d^S,d^S}^S & \Gamma_{d^S,1}^S & \cdots & \Gamma_{d^S,d_{\mathbf{g}}}^S & \Gamma_{d^S,d_{\mathbf{g}}+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_{d^S}^S \end{array}\right] \tag{3}$$

Scheduling matrix Θ is composed of three components:

- Component A is an invertible matrix capturing the relative ordering of iteration vectors. Changing coefficients of this component corresponds to loop interchange, skewing and other *unimodular* transformations.
- Column β reschedules statements statically, at all nesting levels. It expresses code motion, loop fusion and fission.
- Component Γ captures loop shifting (pipelining) effects.

Referring again to Figure 1, multidimensional affine scheduling functions for statements S_1 and S_2 are:

$$\theta^{S_1}(v,h)^T = (0,v,0,h,0)^T$$

$$\theta^{S_2}(v,h,i,j)^T = (0,v,0,h,1,i,0,j,0)^T$$

Those schedules correspond to the execution order of the original loop nest. Note that *odd* positions in the scheduling function are constants (0 and 1 in this example) that correspond to the textual order of the statements, whereas *even* positions correspond to loop counters. For a given values of loop counters v and h the two scheduling vectors are lexicographically equal up to the fourth position. The value at fifth position of the schedule for statement S_1 is 0 and for statement S_2 is 1, meaning that for the same given values of loop counters v and h an instance of statement S_2 is always executed after an instance of statement S_1.

The corresponding scheduling matrix for statement S_1 is:

$$\left[\begin{array}{cc|cc|c} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \cdot \begin{pmatrix} v \\ h \\ N \\ K \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ v \\ 0 \\ h \\ 0 \end{pmatrix}$$

where component A is identity matrix (no rescheduling), the component Γ is zero matrix (schedule does not depend on global parameters) and the β matrix is encoding the static position of the statement inside loop nest.

This decomposed statement scheduling representation format allowed Girbal et al. to enforce invariants upon which long sequences of transformations can be constructed. It also guarantees that any sequence of transformations results in a fully determined sequential schedule [18]. We leverage this property to build a cost model that handles any affine-by-statement transformation sequence.

In this work we concentrate on loop interchange, loop distribution and loop strip-mining. The Graphite[1] framework we use supports the full-fledged $2d^S + 1$ scheduling matrix format, hence we can easily express arbitrary compositions of these transformations. The polyhedral equivalent of interchanging the loop at level k with that at level p consists of interchanging rows k and p in the component A^S of each statement schedule. As we are performing transformations statement-wise, we can easily perform loop interchange of non-perfect nests; statements not deeper than k or p remain unaffected by the transformation.

[1] http://gcc.gnu.org/wiki/Graphite

V. Driving the Optimization Process

To select an optimal vectorization strategy one needs to construct and traverse the relevant search space. We want to do so without generating different syntactic versions of the code and then vectorizing each of them, which is inefficient and sometimes infeasible. Our proposal uses an analytical cost model, and constructs the *finite* optimization search space of chosen loop transformations expressed in terms of modified affine schedules θ'^S and modified iteration domains $\mathcal{D}'^S$ of statements. For each point in the search space we compute an associated cost using a cost function $\phi(\mathbf{x})$.

We model the scheduling of each individual statement independently. Each point in the search space corresponds to a vector $\mathbf{x} = [\theta'^{S_1}, \ldots, \theta'^{S_n}, \mathcal{D}'^{S_1}, \ldots, \mathcal{D}'^{S_n}]$ of modified schedules and domains for each of the n statements of the *SCoP*. The total cost for a given point $\mathbf{x}$ in the space is the sum of costs of executing dynamic instances of all *SCoP* statements according to a new schedule and domain:

$$\phi(\mathbf{x}) = \sum_{i=1}^{n} c(\mathcal{D}'^{S_i}, \theta'^{S_i}). \tag{4}$$

The parameters for the cost function for the single statement S_i are its iteration domain $\mathcal{D}'^{S_i}$ (number of dynamic instances of a statement depends on its iteration domain) and its scheduling θ'^{S_i} (cost of accessing memory by a statement instance depends on execution order of other instances). Section VI describes this cost function in detail.

The optimization goal is to search for a vector of transformations $\mathbf{x}_{min}$ that minimizes the cost function $\phi(\mathbf{x})$:

$$\mathbf{x}_{min} = \min_{\mathbf{x} \in X} \phi(\mathbf{x}) \tag{5}$$

Vector $\mathbf{x}_{min}$ represents an optimal program version in the polyhedral model.

A. Search Space Construction

After extracting the *SCoPs* and building the polyhedral representation of all statements by the Graphite framework, we perform the optimization of each *SCoP* according to Algorithm 1: first we compute the base cost for the unmodified (input program) representation, by computing the cost of executing all dynamic instances of all statements S_i in the original scheduling order. The current optimal cost is stored in $cost_{min}$ and is updated incrementally by applying different transformations (skipping over infeasible ones) on the polyhedral model (stored in vector $\mathbf{x}$) and computing the new costs using cost function $\phi(\mathbf{x})$. Besides the schedule transformation, performed by permuting (PERMUTE) the columns of the component A of the schedule, each possible

level v is strip-mined, which is the way to model the vectorization at level v. At the end of this process the optimal scheduling representation is available in $\mathbf{x}_{min}$.

Note that Algorithm 1 shows only one possible way of constructing a search space. We chose to consider all combinations of loop interchanges due to their impact on vectorization. This small (yet expressive) search space makes it compatible with the constraints of a production compiler.

Algorithm 1 Main optimizing driver

```
d ← level of a deepest statement S in a SCoP
n ← number of statements in a SCoP
{Start with the original schedules and iteration domains}
x_min ← [θ^{S_1}, ..., θ^{S_n}, D^{S_1}, ..., D^{S_n}]
cost_min ← φ(x_min)
for all σ ∈ (set of d-element permutations) do
  for i = 1 to n do
    θ'^{S_i} ← PERMUTE(σ, θ^{S_i})
    d^{S_i} ← level of loop nesting for statement S_i
    for v = 1 to d^{S_i} do
      D'^{S_i} ← STRIPMINE(v, D^{S_i})
      x ← [θ'^{S_1}, ..., θ'^{S_n}, D'^{S_1}, ..., D'^{S_n}]
      if φ(x) < cost_min then
        cost_min ← φ(x); x_min ← x
      end if
    end for
  end for
end for
```

B. Implementation

We implemented the loop nest optimization targeting vectorization within the Graphite framework of the GCC compiler. Polyhedral information is extracted from GIMPLE GCC's intermediate three-address representation. Static control parts are extracted from the SSA-form of GIMPLE. For each statement in a *SCoP* we extract its polyhedral model components: iteration domains $\mathcal{D}^S$, scheduling functions corresponding to original program semantics θ^S, and data access affine subscript functions. We then proceed with search space construction, traversal and cost modeling. The total run time, including exploration of the search space takes at most 0.01s for all loop nests considered.

VI. Polyhedral Modelling of Vectorization Metrics

Several key costs impact the expected performance of vectorized code, including: strides of accesses to memory, memory access alignment, loop trip counts, reduction operations across loop iterations and more. These factors depend on the modified scheduling θ'^S and on the modified iteration domain $\mathcal{D}'^S$ of each statement.

The underlying assumption of vectorization is that the kernel of a loop usually executes faster if vectorized than if not, but that associated overheads may hinder the vectorized version, diminishing its speedup compared to the original scalar version, and more so for loops that iterate a small number of times.

A. Modelling the Access Patterns

Recall from Section IV that access functions for array references are represented as a vector of affine expressions, but there is no notion of the data layout of an array.

One may combine this access function with the data layout of the array. For each array reference, one may form a *linearized memory access function* ℓ, capturing the stream of memory access addresses as a function of the iteration vector:

$$\ell(\mathbf{i}) = b + (\mathrm{L^i}|\mathrm{L^g}|\omega) \times (\mathbf{i}, \mathbf{g}, 1)^T = b + \mathrm{L^i}\mathbf{i} + \mathrm{L^g}\mathbf{g} + \omega \quad (6)$$

where b is the base address[2] of the array and $(\mathrm{L^i}|\mathrm{L^g}|\omega)$ is the row vector of coefficients that encodes the layout information (assuming row-major data layout). This vector is composed of three parts: $\mathrm{L^i}$ is scheduling-dependent, $\mathrm{L^g}$ depends on global parameters, and ω is the constant offset part.

Assuming that matrix `M` is defined as $\mathtt{M}[r_1][r_2]\dots[r_m]$, we can construct the vector R encoding the strides along each subscript[3]. Then the following equation holds:

$$(\mathrm{L^i}|\mathrm{L^g}|\omega) = \mathrm{R} \times \mathrm{F} \quad (7)$$

(recall that matrix F defines the access function f). For example, assuming array `image` is defined as $\mathtt{image}[144][144]$, the linearized access for the array `image` in the statement S_2 of Figure 1 can be represented as:

$$\ell(v, h, i, j) = b + 144v + h + 144i + j.$$

B. Access Pattern Sensitivity to Scheduling

Based on the scheduling matrix representation in Equation (3) the rescheduled time-stamp vector $\mathbf{t}$ is expressed as follows (for modelling purposes we can ignore β):

$$\mathbf{t} = (\mathrm{A}|\Gamma) \times (\mathbf{i}, \mathbf{g})^T = \mathrm{A}\mathbf{i} + \Gamma\mathbf{g} \quad (8)$$

thus the original iteration vector is $\mathbf{i} = \mathrm{A}^{-1}(\mathbf{t} - \Gamma\mathbf{g})$ which together with Equation (6) gives us the new, transformed linearized access function:

$$\ell'(\mathbf{t}) = b + \mathrm{L^i}\mathrm{A}^{-1}\mathbf{t} + (\mathrm{L^g} - \mathrm{L^i}\mathrm{A}^{-1}\Gamma)\mathbf{g} + \omega. \quad (9)$$

Taking as an example the kernel in Figure 1, the linearized access function for array `image` with the original scheduling $\mathbf{t} = \mathbf{i} = (v, h, i, j)$ is:

$$\ell_{\mathtt{image}}(\mathbf{t} = \mathbf{i}) = b + 144v + h + 144i + j$$

[2] b is typically not known at compilation time; nevertheless, we are only interested in its alignment modulo the VF, which is generally available.

[3] $\mathrm{R} = (\prod_{i=1}^{m-1} r_i, \prod_{i=2}^{m-1} r_i, \dots, \prod_{i=m-1}^{m-1} r_i, 1)$.

After interchanging levels 3 and 4 (expressed as a transformation on component A of the schedule) the new access function become:

$$\ell_{\texttt{image}}(\mathbf{t}=(v,h,j,i)) = b + 144v + h + i + 144j$$

Notice that strides with respect to the new scheduling dimensions have changed. This has dramatic impact on the performance of the vectorized code. Indeed, if we chose to vectorize the level 3 (which now corresponds to original loop j) the vectorized code will suffer from a very costly memory access operations: the stride is 144, so the elements of a vector cannot be loaded in one vector-load instruction.

C. Cost Model Function

Our cost model is based on modelling the total execution time of all statement instances, given the modified iteration domain $\mathcal{D}'^S$ and the modified schedule θ'^S of each statement S. We compute the cost function for [4] statement S as follows:

$$\begin{aligned} c(\mathcal{D}'^S, \theta'^S) \quad = \quad & \frac{|\mathcal{D}'^S|}{\text{VF}}(\sum c_{\text{vect\_instr}}) + \\ & \sum_{m\in(\mathcal{W}_S)} (c_a + \frac{|\mathcal{D}'^S|}{\text{VF}}(c_{\text{vect\_store}} + f_m)) + \\ & \sum_{m\in(\mathcal{R}_S)} (c_a + \frac{|\mathcal{D}'^S|}{\text{VF}}(c_{\text{vect\_load}} + c_s + f_m)) \end{aligned}$$

where $|\mathcal{D}'^S|$ denotes the integer cardinality of the iteration space (total number of dynamic instances of S) and VF is the vectorization factor. Factor c_s considers the penalty of load instructions accessing memory addresses with a *stride* across the loop being vectorized. The memory access stride δ_d w.r.t. a schedule dimension d can be determined directly from vector L^{i} by looking at its d-th component. Vector registers of SIMD architectures can be directly loaded with consecutive memory addresses only. Accesses to non-unit strided addresses require additional data unpack and/or pack operations [13].

More precisely, loading VF memory addresses with a stride δ_{d_v} across the loop being vectorized may require δ_{d_v} vector loads (each with cost c_1), followed by $\delta_{d_\text{v}} - 1$ vector extract odd or extract even instructions (each with cost c_2), to produce just one vector register holding the desired VF elements. If $\delta_{d_\text{v}} = 0$, then the same value needs to be replicated to fill a vector register using a "splat" instruction (with cost c_0). Factor c_s is thus defined as follows:

$$c_s = \left\{ \begin{array}{ll} c_0 & : \delta_{d_\text{v}} = 0 \\ 0 & : \delta_{d_\text{v}} = 1 \\ \delta_{d_\text{v}} \cdot c_1 + (\delta_{d_\text{v}} - 1)\cdot c_2 & : \delta_{d_\text{v}} > 1 \end{array} \right\} \qquad (10)$$

[4]please note that this cost function applies only if $d^S > \delta_{d_\text{v}}$ is satisfied, i.e., the statement is nested within vectorized level. Please also note that this cost function, for lack of space, is simplified with respect to the cost function that is implemented into GCC – it does not show the reduction costs and the vector store with strides for example.

Factor c_a considers the *alignment* of loads and stores. Typically, accesses to memory addresses that are aligned on VF-element-boundaries are supported very efficiently, whereas other accesses may require loading two aligned vectors from which the desired unaligned VF elements are extracted (for loading) or inserted (for storing). This alignment overhead may be reduced considerably if the stride δ of memory addresses accessed across loop dimensions[5] $d_\text{v}+1, \ldots, d^S$ is a multiple of VF, because then the misalignment offset becomes invariant w.r.t. the vectorized loop. In these cases there is the opportunity to reuse loaded vectors and use invariant extraction masks.

It is easy to check if the misalignment inside the vectorized loop is invariant by considering the transformed linearized access function:

$$\ell(\mathbf{i}) = b + \mathrm{L}^{\mathrm{i}}_1 i_1 + \ldots + \mathrm{L}^{\mathrm{i}}_{d_\text{v}} i_{d_\text{v}} + \cdots + \mathrm{L}^{\mathrm{i}}_{d^S} i_{d^S} + \mathrm{L}^{\mathrm{g}}\mathbf{g} + \omega \quad (11)$$

— all coefficients from $\mathrm{L}^{\mathrm{i}}_{d_\text{v}+1}$ to $\mathrm{L}^{\mathrm{i}}_{d^S}$ (corresponding to strides of all inner loops of the vectorized loop) have to be divisible by VF.

If the misalignment is invariant in the vectorized loop we also check if the base address accessed on each first iteration of the vectorized loop (d_v) is known to be aligned on VF-element-boundary; if so there is no need for re-aligning any data: $c_a = 0$. This is done by considering initial alignment properties and strides across outer-loops (enclosing the vectorized loop). The strides of enclosing outer loops maintain alignment invariance if coefficients $\mathrm{L}^{\mathrm{i}}_1, \mathrm{L}^{\mathrm{i}}_2, \ldots, \mathrm{L}^{\mathrm{i}}_{d_\text{v}-1}$ are all divisible by VF.

Putting all considerations for alignment together, the alignment cost can be modelled as:

$$c_a = \left\{ \begin{array}{lll} 0 & : & \text{aligned} \\ |\mathcal{D}^S|(c_1 + c_3 + c_4) & : & \text{var. misalign.} \\ |\mathcal{D}^S_{1..d_\text{v}-1}|(c_1 + c_3) + & : & \text{fixed misalign.} \\ |\mathcal{D}^S|(c_1 + c_4) & & \end{array} \right\} \qquad (12)$$

where c_3 represents the cost of building a mask based on the misalignment amount, c_4 is the cost of extraction or insertion and c_1 is the vector load cost. $|\mathcal{D}^S_{1..d_\text{v}-1}|$ denotes the number of iterations around the vectorized loop level.

The vectorization factor VF of a loop is determined according to the size of the underlying vector registers and the smallest data-type size operated on inside the loop. Each individual vector register will thus hold VF values of this small size. However, if there are variables in the loop of larger size, storing VF copies of them will require multiple vector registers, which in turn implies that the associated instructions need to be replicated. Factor f_m records the extra overhead associated with this replication. Additional factors that depend on specific machine resources available may also impact the performance of vectorization, such as

[5]Dimensions correspond to loops nested in a vectorized loop

the size of register files, available ILP, and complex vector instructions.

To summarize, the above vectorization profitability metrics can be classified into three classes: (1) **Scheduling invariant** metrics: these are not affected by changes to the execution order of statement instances. The cost of vector to scalar reduction and of multiple type support belong to this category. (2) **Scheduling sensitive** metrics: these are affected by changes to the execution order of statement instances. Costs associated with strided and unaligned memory accesses, as well as spatial locality, belong to this category. (3) **Code generation dependent** metrics: these depend on the actual code-generation strategy implemented in a compiler. Costs related to idiom recognition belong to this category.

Our approach of integrating the cost model within the polyhedral framework handles the first two categories quite naturally, and is especially powerful in dealing with the second category whose metrics vary with changes to the scheduling order, and are thus affected by loop transformations such as interchange. Category 3 is less suitable for our design, because it relies on generating the compiler's internal structures from the modified polyhedral representation; yet no significant performance factor belongs to this category, according to our experiments. The metrics that have the greatest impact are those that depend on the scheduling function θ^S, which are well handled by our model.

VII. Experimental Results

	N_1	N_2	N_3	N_4	δ_1	δ_2	δ_3	δ_4
interp_fp	512	16			1,2	1,0,2		
interp	512	16			1,2	1,0,2		
bkfir	512	32			1,0	1,1		
dct	8,8	8,8	8,8		8,0	0,1	1,8	
convolve	128	128	16	16	128, 0, 128	1, 0, 1	128, 16, 0	1, 1, 0
H264	12,7	12,7			1	1		
dissolve	128	128			1	128		
alvinn	512,32	32,32			1,1	512,512		
MMM	16	16	16		16,0	0,1	1,16	
MMM^T	16	16	16		16,0	0,1	1,16	

Table I
Benchmarks

We evaluate our approach by introducing our model into the polyhedral framework of GCC and comparing its performance estimates for different loop interchanges and vectorization alternatives against actual execution runs of a set of benchmarks. Table I summarizes the main relevant features of the kernels used in our experiments: a rate 2 interpolation (*interp*), block finite impulse response filter (*bkfir*), an 8×8 discrete cosine transform (*dct* [23]), 2D-convolution (*convolve*), a kernel from H.264 (*H264*), video image dissolve (*dissolve*), weight-update for neural-nets training (*alvinn*) and a 16×16 matrix-matrix multiply (*MMM*) (including a transposed version *MMM*^{T}).

The first four columns of Table I show the number of iterations N_i of loops nested within the main loop-nest of each benchmark, starting with N_1 for the outermost loop and moving inwards. Loop nests with less that 4 nested loops have empty entries (e.g., N_2 refers to the innermost loop in the doubly-nested *bkfir*). Multiple values in an entry represent multiple distinct loop nests.

Similarly, the next four columns of Table I show the strides δ_i of the memory references across each of the nested loops, with multiple values in an entry representing the strides of different memory references. For example, strides of 8, 512 and 16 are found in the innermost loops of *dct, alvinn* and *MMM* respectively, where columns of 2D arrays are scanned resulting in strides at the length of the rows. Lastly, zero strides imply that duplication of a single value across a vector is required.

We first evaluate the cost-model qualitatively, demonstrating that the scores it computes are consistent using one detailed example (subsection VII-A). The following subsection evaluates the model relative to actual experiments on a set of kernels, analyzing the mispredictions and showing that overall the relative performance ordering of the variants is largely preserved. Finally subsection VII-C demonstrates the impact of using a polyhedral-model guided vectorization compared to using one of three default vectorization schemes: innermost loop vectorization, innermosting (permuting a loop to the innermost position and vectorizing it there), and in-place outer-loop vectorization with realignment optimization. We show that no single approach is best for all benchmarks, hence the need for a sophisticated cost model.

A. Qualitative Evaluation

We use the *convolve* kernel qualitatively (see Figure 1, Section III). For lack of space, we study only a small subset of the search space described in Section III and Algorithm 1, restricting our attention to the $d \times d$ combinations of shifting each loop inwards/outwards and vectorizing it there, plus the option to vectorize each of the d loops without any interchange. Note however that our technique opens up a much larger (polyhedral) transformation space and the driver described in Section V can compute scores for all $d(d!)$ combinations, if desired.

The results of running our model against the $d \times d = 16$ combinations, estimating the performance of each combination for a Cell/SPU and a PowerPC system are shown in Table II and Table III respectively. The loops are numbered in the tables from 1 (outer-most) to 4 (inner-most). Entry (i, j) shows the estimated speedup over the sequential version, obtained by shifting loop j to position i followed by vectorizing loop j at new position i. Thus entries along the diagonal refer to vectorization with no interchange. Entries (4,4), (4,2), (3,2) (in bold) correspond to the vectorization alternatives shown in Figures 2a, 2b, 2c respectively.

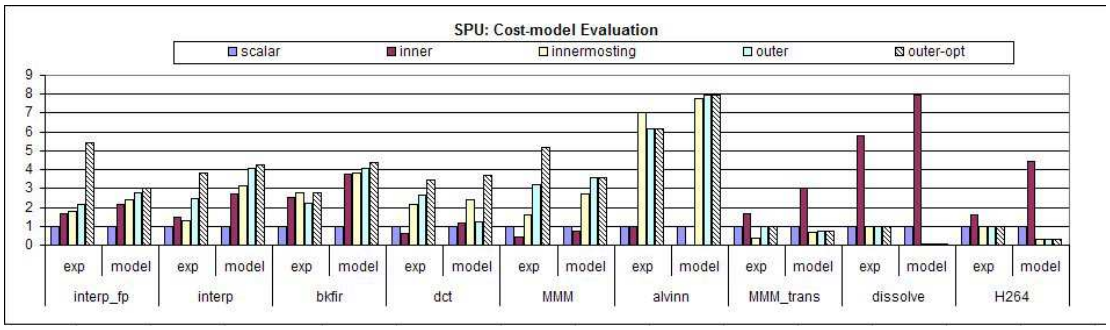


Figure 3. Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on the Cell SPU

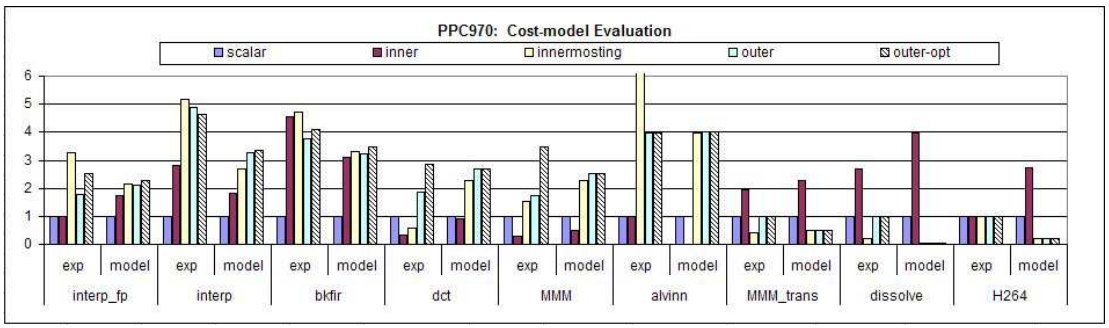


Figure 4. Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on PPC970

loop-level	1(v)	2(h)	3(i)	4(j)
1	0.26	4.00	0.24	4.00
2	0.26	4.06	0.24	4.21
3	0.26	**4.34**	0.23	4.56
4	0.27	**3.76**	0.24	**3.72**

Table II
CONVOLVE: SPU ESTIMATED SPEEDUP FACTORS

loop-level	1(v)	2(h)	3(i)	4(j)
1	0.21	3.21	0.19	3.21
2	0.21	3.21	0.19	3.38
3	0.21	**3.18**	0.19	3.70
4	0.21	**3.37**	0.20	**2.99**

Table III
CONVOLVE: PPC ESTIMATED SPEEDUP FACTORS

The convolve entry in Table I reveals the key factor for the performance degradations predicted for loops v, i (columns 1 and 3) — there are very large strides along these loops (δ_1, $\delta_3 = 128$). The overhead involved in vectorizing these loops and strides is described in Section VI. The remaining candidate loops for vectorization are therefore loops 2 and 4 (h and j). The best speedup is predicted for entry (3, 4) which corresponds to using outer-loop vectorization to vectorize the *j*-loop after shifting it to level 3. The original *i*-nest is a perfect nest (there are no operations outside the innermost loop within that nest) and so there are no overheads incurred by this interchange (as opposed to interchanging an imperfect-nest like the *h*-loop, e.g. as in cases (4,2),(3,2)/Figures 2b,2c, which involve scalar expansion and loop-distribution costs). In addition, outer-loop vectorization avoids reduction-epilog costs and also increases the portion of the code that is being vectorized compared to vectorizing the *j*-loop in its original innermost location. Note that this choice is different from the traditional approach: compilers usually either apply inner-most loop vectorization (entry $(4, 4)$ in the tables) or apply innermosting (entries $(4, *)$).

Partial experimental evaluation of *convolve* confirms these predictions. In Figure 5 we show the obtained speedups relatively to the cost model estimations (denoted *exp*,*model* respectively) for PPC970 and Cell SPU for entries $(3, 2)$, $(4, 2)$, $(3, 4)$ and $(4, 4)$ in the tables. The relative ordering of the speedups for both platforms is accurate [6] and the cost model is able to identify the best choice among the three.

B. Experimental Evaluation

We now validate quantitatively the estimates produced by the cost model. For each benchmark we report two sets of results: one showing the experimentally observed speedups, and the other showing estimated speedups computed by the cost model (denoted *exp*,*model* respectively in Figures 3, 4). When a given vectorization technique cannot be applied due to limitations of our current implementation of vectorization in the GCC compiler, the scalar performance is reported. This happens in some cases of strided accesses that are

[6]The low 1.44x measured speedup on the Cell SPU for alternative $(4, 2)$ (corresponding to Figure 2b) is due to an aliasing bug in GCC that results in bad scheduling. The out-of-order wide-issue (5 slots) PowerPC970 is less sensitive to this, but on the in-order 2-width-issue SPU performance drastically suffers as a result. The cost model obviously cannot (and should not) predict compiler bugs, however it can, as in this case, help reveal them.

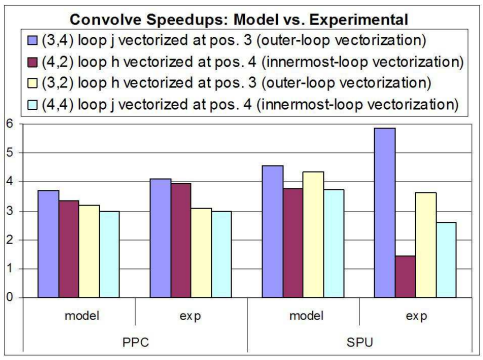


Figure 5. Cost model evaluation: comparison of predicted and actual impact for convolve kernel on PPC970 and Cell SPU

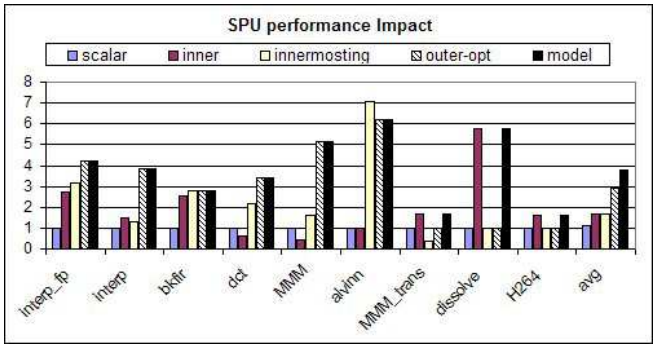


Figure 6. Predicted optimal vs. 4 fixed strategies on SPU

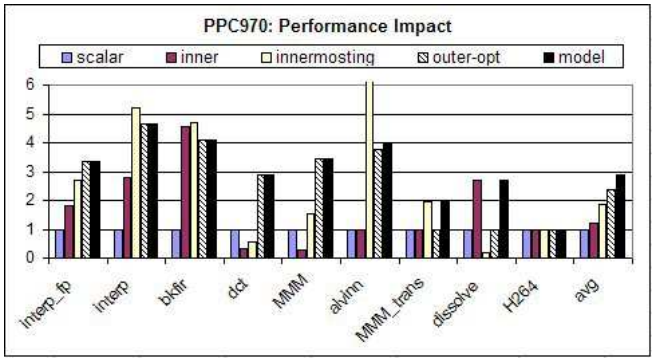


Figure 7. Predicted optimal vs. 4 fixed strategies on PPC

not yet fully supported (and that would certainly degrade performance).

We evaluate the relative speedup of four different vectorization alternatives: innermost-loop vectorization (*inner*), interchange followed by innermost-loop vectorization (*innermosting*), and in-place outer-loop vectorization, with and without optimized realignment using unrolling (*outer* and *outer-opt*).

The experiments were generated automatically using an enhanced version of GCC. Speedup are measured over the sequential version of the benchmark, compiled with the same optimization flags. Interchange, when used, was applied manually. Time is measured using the `getrusage` routine on powerpc970, and the decrementer utility on the SPU. Experiments were performed on the IBM PowerPC PPC970 processor with Altivec, and an SPU of the Cell Broadband Engine. Both architectures have 128 bit wide vector registers, and similar explicit alignment constraints.

The first set of kernels (interp, bkfir, dct and MMM) is expected to gain most from in-place outer-loop vectorization with realignment optimization, as they consist of imperfect loop-nests (and therefore get penalized for interchange), and exhibit high data-reuse opportunities across the (vectorized) inner-loop that can be exploited by the unrolling optimization. They also have inner-loop reductions (which are generally done more efficiently using outer-loop vectorization), and two of the benchmarks in this set (dct and MMM) also have large strides in the innermost loop (as the access is column-wise). Alvinn has a perfect nest and no reuse opportunities, and therefore in-place outer-loop vectorization should not gain over traditional interchange, but innermost loop vectorization should be avoided due to the large stride. The last group of benchmarks (MMM^T, dissolve and H264) have consecutive access in the innermost loop, but strided access in the outer-loop, and so for these we expect inner-loop vectorization to be the best technique.

This behavior can be clearly observed in the SPU speedups in Figure 3, where overall the *exp* and *model* graphs are largely consistent, with the preservation of the relative performance ordering of the variants. Exceptions are due to low-level target-specific factors that our model does not take into account. Most notable is the misprediction in the first set of benchmarks, where *bkfir* and *dct* are the only benchmarks for which outer-loop vectorization is inferior to innermost loop vectorization due to an SPU specific issue (unhinted branch).

Target-specific issues come to play also on the PPC970 (Figure 4). The most significant one appears in the fixed-point bkfir and interp where inner-loop vectorization employs a specialized Altivec instruction to compute a dot-product pattern. We have not yet incorporated idioms into the cost-model and so it does not anticipate this behavior. The model also does not try to estimate register pressure, and therefore does not predict the degradation in performance incurred by the unrolling optimization on interp due to register spilling (this problem does not occur for SPU having 128 vector registers, compared to the 32 Altivec registers of PowerPC970). Lastly, in some cases interchange can be done with smarter scalar-expansion (hoisting), whereas the model estimates the associated overhead of a naive scheme. This sometimes pessimizes the predicted speedup of interchanged versions both on PPC and the SPU.

C. Performance Impact

Lastly, we evaluate the overall benefit of using a cost-model for loop-nest vectorization. In Figures 6, 7 we compare the speedups obtained by the following four fixed strate-

gies: (1) always leave the loops sequential, (2) always apply innermost loop vectorization, (3) always apply innermosting (followed by innermost loop vectorization), and (4) always apply optimized outer-loop vectorization. This is compared to the 5th and last strategy which is to pick the vectorization scheme recommended by the cost model. The last set of bars in each figure shows the geometric mean of the speedups obtained by each of these 5 strategies across all benchmarks.

On the SPU, in all but one case (alvinn) the model correctly predicted the best vectorization technique. Using the cost-model driven approach, we obtain an average speedup factor of 3.5 over the scalar version, which is an improvement of 36% over the optimized in-place outer-loop vectorization technique, and 2.3 times faster than the innermost vectorization approach, on average.

On the PPC970, the cost model mispredicts in 3 cases (interp, bkfir and alvinn). The Overall speedup factor obtained by the cost-model driven approach is 2.9 over the scalar version, an improvement of 50% over outer-opt, and 2.3 times faster than innermost loop vectorization, on average.

VIII. Conclusions

We presented a cost model and a loop transformation framework to extract subword parallelism opportunities and to select an optimal strategy among them. This framework is based on polyhedral compilation, leveraging its representation of memory access patterns and data dependences as well as its expressiveness in building complex sequences of transformations. The main factors contributing to the profitability of vectorization can be captured by the polyhedral representation itself, alleviating the cost of code generation when iteratively searching for an optimal vectorization strategy. The framework is implemented in GCC 4.4 and was validated on a number of representative loop nests and on multiple architectures with slightly different SIMD computation capabilities.

Acknowledgment

This research is supported by the SARC, ACOTES and HiPEAC European grants. Part of the work was done while the first author visited the IBM Haifa Research Lab on HiPEAC internship. We would also like to thank Sebastian Pop, AMD and other contributors of the Graphite project.

References

[1] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An integrated Simdization framework using virtual vectors," in *ICS*, 2005.

[2] A. J. C. Bik, *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the Intel architecture," *IJPP*, vol. 30, no. 2, pp. 65–98, 2002.

[4] D. Nuzman and A. Zaks, "Autovectorization in GCC – two years later," in *the GCC Developer's summit*, June 2006.

[5] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *CGO*, March 2005.

[6] D. Nuzman and A. Zaks, "Outer-loop vectorization - revisited for short SIMD architectures," in *PACT*, October 2008.

[7] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelization and locality optimization system," in *PLDI*, Jun. 2008.

[9] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *PLDI*, Jun. 2008.

[10] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Tr. on Prog. Lang. and Systems*, vol. 9, no. 4, pp. 491–542, 1987.

[11] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[12] J. Shin, J. Chame, and M. W. Hall, "Compiler-controlled caching in superword register files for multimedia extension architectures," in *PACT*, September 2002.

[13] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," in *PLDI*, 2006.

[14] P. Boulet, A. Darte, T. Risset, and Y. Robert, "(Pen)-ultimate tiling?" in *IEEE Scalable High-Performance Computing Conf.*, May 1994.

[15] C. Cascaval, L. Derose, D. A. Padua, and D. A. Reed, "Compile-time based performance prediction," in *LCPC*, 1999.

[16] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Probabilistic miss equations: Evaluating memory hierarchy performance," *IEEE Trans. Comput.*, vol. 52, no. 3, pp. 321–336, 2003.

[17] P. Feautrier, "Array expansion," in *ICS*, St. Malo, France, Jul. 1988.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Intl. J. of Parallel Programming*, vol. 34, no. 3, pp. 261–317, Jun. 2006, special issue on Microgrids.

[19] P. Feautrier, "Some efficient solutions to the affine scheduling problem, part II, multidimensional time," *Intl. J. of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992, see also Part I, one dimensional time, 21(5):315–348.

[20] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," University of Maryland, Tech. Rep. CS-TR-3193, 1993.

[21] A. Lim and M. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *PoPL'24*, Paris, Jan. 1997, pp. 201–214.

[22] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT*, Sep. 2004.

[23] C. G. Lee, "UTDSP benchmarks," http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 1998.